Why Add Versions To Go?

Go & Versioning, Part 10

Russ Cox

June 7, 2018

research.swtch.com/vgo-why-versions

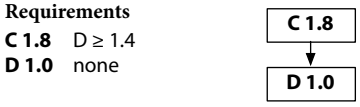
People sometimes ask me why we should add package versions to Go at all. Isn't Go doing well enough without versions? Usually these people have had a bad experience with versions in another language, and they associate versions with breaking changes. In this post, I want to talk a little about why we do need to add support for package versions to Go. Later posts will address why we won't encourage breaking changes.

The `go get` command has two failure modes caused by ignorance of versions: it can use code that is too old, and it can use code that is too new. For example, suppose we want to use a package D, so we run `go get D` with no packages installed yet. The `go get` command will download the latest copy of D (whatever `git clone` brings down), which builds successfully. To make our discussion easier, let's call that D version 1.0 and keep D's dependency requirements in mind (and in our diagrams). But remember that while we understand the idea of versions and dependency requirements, `go get` does not.

```
$ go get D
```

Now suppose that a month later, we want to use C, which happens to import D. We run `go get C`. The `go get` command downloads the latest copy of C, which happens to be C 1.8 and imports D. Since `go get` already has a downloaded copy of D, it uses that one instead of incurring the cost of a fresh download. Unfortunately, the build of C fails: C is using a new feature from D introduced in D 1.4, and `go get` is reusing D 1.0. The code is too old.

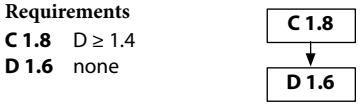
```
$ go get C
```



broken!

Next we try running `go get -u`, which downloads the latest copy of all the code involved, including code already downloaded.

```
$ go get -u C
```



(C 1.8 was tested with D 1.4 and has an unexpected incompatibility with D 1.6.)

broken!

Unfortunately, D 1.6 was released an hour ago and contains a bug that breaks C. Now the code is too new. Watching this play out from above, we know what `go get` needs to do: use D ≥ 1.4 but not D 1.6, so maybe D 1.4 or D 1.5. It's very difficult to tell `go get` that today, since it doesn't understand the concept of a package version.

Getting back to the original question in the post, *why add versions to Go?*

Because agreeing on a versioning system—a syntax for version identifiers, along with rules for how to order and interpret them—establishes a way for us to communicate more precisely with our tools, and with each other, about which copy of a package we mean. Versioning matters for correct builds, as we just saw, but it enables other interesting tools too.

For example, the obvious next step is to be able to list which versions of a package are being used in a given build and whether any of them have updates available. Generalizing that, it would be useful to have a tool that examines a list of builds, perhaps all the targets built at a given company, and assembles the same list. Such a list of versions can then feed into compliance checks, queries into bug databases, and so on. Embedding the version list in a built binary would even allow a program to make these checks on its own behalf while it runs. These all exist for other systems already, of course: I'm not claiming the ideas are novel. The point is that establishing agreement on a versioning system enables all these tools, which can even be built outside the language toolchain.

We can also move from query tools, which tell you about your code, to development tools, which update it for you. For example, an obvious next step is a tool to update a package's dependencies to their latest versions automatically whenever the package's tests and those of its dependencies continue to pass. Being able to describe versions might also enable tools that apply code cleanups. For example, having versions would let us write instructions "when using D version ≥ 1.4, replace the common client code idiom `x.Foo(1).Bar(2)` with `x.FooBar()`" that a tool like `go fix` could execute.

The goal of our work adding versions to the core Go toolchain—or, more generally, adding them to the shared working vocabulary of both Go developers and our tools—is to establish a foundation that helps with core issues like building working programs but also enables interesting external tools like these, and certainly others we haven't imagined yet.

If we're building a foundation for other tools, we should aim to make that foundation as versatile, strong, and robust as possible, to enable as many other tools as possible, with as little hindrance as possible to those tools. We're not just writing a single tool. We're defining the way all these tools will work together. This foundation is an API in the broad sense of something that programs must be written against. Like in any API, we want to choose a design that is powerful enough to enable many uses but at the same time simple, reliable, consistent, coherent, and predictable. Future posts will explore how vgo's design decisions aim for those properties.